CSCE 2014 – Programming Project 4

Midpoint Due Date – 7/13/17 at 11:59 PM
Final Due Date – 7/20/17 at 11:59 PM

1. Problem Statement:

In this project, you will use several of the data structures we've been studying so far, including vectors and stacks, to construct a rudimentary mathematical expression calculator. You will write a program that allows a user to enter an expression, such as **(3.2 + 5) / 7 * 2.1**, and then evaluates it. In this case, the program would produce the result **2.46**.

1.1 Infix, Prefix, and Postfix Expressions

Expressions like the one used in the example above are called "**infix**", meaning operators (e.g. 3.2, 5, 7, 2.1) are usually placed on either side of an operand (e.g. +, -, *, /). Although this system is typically used when humans perform arithmetic, it does have several important flaws that make working with them difficult for computers.

One of the most important is that infix expressions are potentially ambiguous. For example, "**5 + 3 * 4**", could be interpreted as either "**(5 + 3) * 4**" or as "**5 + (3 * 4)**". To avoid ambiguity, scholars and mathematicians have collectively agreed to assign precedence levels to each operator. For example, multiplication and division have higher precedence than addition and subtraction, and exponentiation has a higher precedence than multiplication or division. These precedence rules allow us to correctly interpret the expression, meaning "**5 + (3 * 4)**" would be chosen over "**(5 + 3) * 4**", given no other information. Of course, if we ever want (or need) to be more explicit, we can also use parentheses to resolve ambiguity issues.

There do exist other notation systems besides infix. The next two most common are called "**prefix**" and "**postfix**". In the prefix system, operators are placed **before** the operands, and in the postfix system, operators are placed **after** the operands. As a simple example, "**3 + 5**" becomes "**+ 3 5**" in the prefix system and "**3 5 +**" in the postfix system.

Both the prefix and postfix systems have the nice property of being **completely unambiguous**. The infix expression "**5 + (3 * 2)**" becomes "**+ 5 * 3 2**" in the prefix system and "**5 3 2 * +**" in the postfix system, while the expression "**(5 + 3) * 2**" becomes "*** + 5 3 2**" and "**5 3 + 2 ***", respectively. There is no need for parentheses in the postfix expression because there is no possibility for misinterpretation. This property makes evaluating prefix and postfix expressions *considerably* easier than evaluating infix ones. **Our general strategy, then, will be to convert the infix expression to an equivalent prefix or postfix expression that can then be evaluated directly.**

1.2 Evaluating Prefix and Postfix Expressions

Functionally, there is little difference between working with prefix expressions and working with postfix ones. Historically, however, computer scientists have dealt more with postfix expressions, since the associated algorithms are marginally simpler to implement. We will follow that tradition as well, so from now on, we will limit our discussion to postfix expressions.

Postfix expressions can be efficiently evaluated using a simple stack-based algorithm. To help demonstrate how it works, we will use one of the expressions above as an illustrative example. The infix expression "**(5 + 3) * 2**" corresponds to the postfix expression "**5 3 + 2 ***". It should be apparent from the infix expression that the result is 8 * 2 = **16**.

We will iterate over the postfix expression from **left to right**. Every time we see a **number**, we will **push** that number onto a stack. Every time we see an **operator**, we will **pop two numbers** from the stack, perform the associated operation, and then **push** the result back on the stack. When we reach the end of the expression, there should only be one number on the stack, which will be the result of the expression. This is a detailed example:

Token	Stack	Notes
5	5	Numbers are pushed directly onto the stack
3	5 3	Numbers are pushed directly onto the stack
+	~~5 3~~	1. Pop two numbers off the stack
	~~5 3~~	2. Add the numbers together (because the token is a +)
	~~5 3~~ 8	3. Push the result onto the stack
2	8 2	Numbers are pushed directly onto the stack
*	~~8 2~~	1. Pop two numbers off the stack
	~~8 2~~	2. Multiply the two numbers (because the token is a *)
	~~8 2~~ 16	3. Push the result onto the stack
	16	DONE – The result is at the top of the stack.

It is quite easy to check if a given postfix expression is well-formed while this algorithm is executing, as there are just two possible erroneous conditions we might encounter:

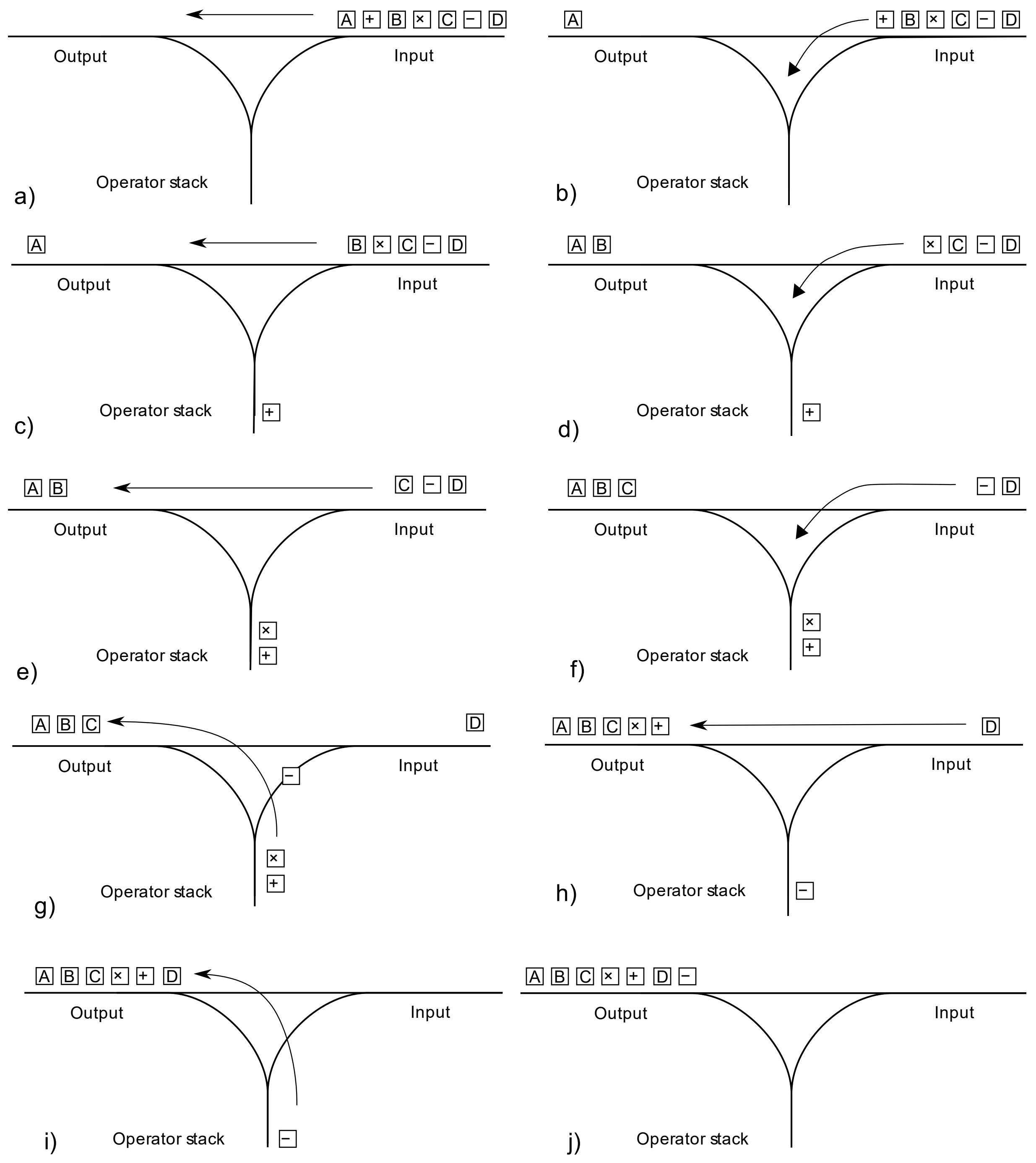
1. If at any point, we try to pop from an empty stack, the expression must have been invalid.
2. If the stack is either empty or contains more than 1 element *after the algorithm has completed*, the expression is also invalid.

1. 3 Dijkstra's Shunting Yard

We've seen that postfix expressions are very simple to evaluate, but we still need to handle the conversion from infix to postfix in the first place. To accomplish that, we need an algorithm called Dijkstra's Shunting Yard. The image below[1] demonstrates the basic process.

We envision our input expression as a series of train cars, one car per token, lying on a railroad track. Each car has the option of either moving forward on the same track or being rerouted to an auxiliary track below. The auxiliary track leads to a dead-end, but it also connects back to the original track, acting as a physically-inspired stack. Shunting yards allow train operators to adjust the position of cars within the train, but we will use the same concept to adjust the position of operators and operands in a mathematical expression.

[1] https://en.wikipedia.org/wiki/Shunting-yard_algorithm



This is the algorithm in detail:

We will process each token from the original infix expression one a time, using a stack to simulate the auxiliary shunting yard track. Our logic is specialized based on each type of token.

- **Numbers –** Numbers pass through the yard without modification. They go directly to the output.
- **Operators –** Operators are rerouted through the auxiliary track, but we must deal with the fact that the operators have different precedence levels. All operators with precedence ≥ the incoming operator must first be pushed from the auxiliary track onto the output before the incoming operator can be placed in the auxiliary track.

- **Left parentheses** – Parentheses allow us to manually intervene when the traditional precedence rules do not have the desired effect. Left parentheses are pushed directly onto the auxiliary track.
- **Right parentheses** – Right parentheses have a slightly more complicated logic associated with them. We must continually pop cars from the auxiliary track onto the output until we find the matching left parenthesis in the auxiliary track. Once we find it, we can remove that car completely. (Remember, postfix expressions do not have parentheses at all because they are completely unambiguous.)

After every token in the input expression has been processed, any remaining cars in the auxiliary track (items on the stack) are pushed onto the output.

As with the postfix evaluation algorithm, it is possible for the Shunting Yard algorithm to fail. We primarily concern ourselves with **mismatched parentheses**. It is possible for an expression to be missing either a left parenthesis or a right one, so we must address both possibilities.

We can detect a missing left parenthesis when the current token is a right parenthesis and we are popping operators from the stack. If the stack becomes empty before a match is found, the expression was invalid. Similarly, we can detect a missing right parenthesis by examining the operators that remain in the stack as they are being popped onto the output. If any of them are a left parenthesis, the expression was also invalid.

A detailed example of an execution of the Shunting Yard algorithm is provided below, for the expression: "**3 * (4 / 5) + 2**":

Token	Output	Stack	Notes
3	3		Numbers pass straight through the yard
*	3	*	Operators are pushed onto the stack
(	3	* (	Left parentheses are pushed onto the stack
4	3 4	* (	Numbers pass straight through the yard
/	3 4	* (/	Operators are pushed onto the stack
5	3 4 5	* (/	Numbers pass straight through the yard
)	3 4 5 /	*	Pop from the stack until the matching '(' is found. Then pop the '('.
+	3 4 5 / *		1. '*' has a higher precedence than '+', so it must be popped first
	3 4 5 / *	+	2. '+' can safely be pushed onto the stack
2	3 4 5 / * 2	+	Numbers pass straight through the yard
	3 4 5 / * 2 +		The stack is emptied onto the output.
	3 4 5 / * 2 +		DONE!

2. Design:

There are three fundamental design issues: the driver, the shunting yard algorithm, and the postfix evaluation algorithm.

2.1 Driver

Our driver will be very simple. It should:

1. Ask the user to enter an infix expression.
2. Print the program's interpretation of the infix expression.
3. Attempt to convert the infix expression to an equivalent postfix expression. (If the expression is invalid, inform the user and exit the program.)
4. Attempt to evaluate the postfix expression. (If the postfix expression is invalid, inform the user and exit the program.)
5. Display the result of the evaluation.

2.2 Converting Infix to Postfix

To convert the infix expression given by the user to a postfix expression, you will need to implement Dijkstra's Shunting Yard algorithm. Doing so becomes complicated as more operators are added, so we will try to reduce that complexity by limiting ourselves to just 4:

Operator	Symbol	Priority
Addition	+	1
Subtraction	-	1
Multiplication	*	2
Division	/	2

Furthermore, we will allow users to enter parentheses to explicitly indicate in which order certain operations should be performed.

Interestingly, neither the actual value of the operands nor the meaning behind the operator symbols is relevant when implementing the Shunting Yard algorithm. We just need to know whether a certain string represents an operator and if so, what priority that operator has. In that regard, the Shunting Yard algorithm can be viewed as a simple transformation routine—it rearranges an expression, but it does not actually perform any mathematical operations.

2.3 Evaluating a Postfix Expression

If the Shunting Yard algorithm succeeds in converting the user's infix expression to an equivalent postfix expression, the next step will be to evaluate that postfix expression using the algorithm outlined in the last section.

Unlike the Shunting Yard implementation, the evaluation implementation DOES need to be aware of the intended mathematical interpretation of a given symbol (e.g. "+" means addition, etc.), and the actual value of the operand is important. It's important that you understand conceptually the difference between the two algorithms. The first transforms one expression into another expression, while the second transforms an expression into a result (a number). Both use stacks as part of their implementation, and both need to be able to decide whether a given string represents an operator, but when writing the evaluation implementation, **you will need to translate a string into an action or into a number.**

You have the knowledge necessary to use templates in the implementation of the evaluation routine (so we could work with integers, doubles, floats, etc.), but doing so is not likely to be particularly helpful. Most calculators generally assume that all operands are real numbers, so it is alright for us to assume that **all numbers are doubles** in this assignment.

3. Implementation:

Driver

We will use vectors of strings to represent individual expressions (both infix and prefix). Each element of the vector will represent a single token. For example, the expression "**3.2 * (4.0 / 5.1) + 2**" would be represented as a vector containing **{"3.2", "*", "(", "4.0", "/", "5.1", ")", "+", "2"}**.

One of the first issues you must address is how to read the input from the user into an empty vector. Generally, parsing expressions that may or may not have spaces in them is a difficult problem, and doing so is not the purpose of this assignment, so we will take a simpler approach and assume that the user inserts spaces between each term of the expression. For example, we can assume the user enters "**3.2 * (4.0 / 5.1) + 2**" (with the spaces) instead of "**3.2*(4.0/5.1)+2**" (without spaces). If that is the case, the expression can easily be parsed using the **>>** operator:

```
Vector<string> expression;
string token;
while (cin >> token)
{
    expression.pushBack(token);
}
```

Remember the **>>** operator skips over spaces automatically and returns false when it fails. With **cin**, the stream operator will fail if the user enters either **Ctrl + D** or **Ctrl + Z**, depending on your operating system.

Shunting Yard Implementation

The next issue is how to implement the Shunting Yard algorithm. For this part, you will need to implement the following function prototype:

```
bool shuntingYard(const Vector<string>& expression, Vector<string>& postfix)
```

This function will be given the infix expression provided by the user and will attempt to convert it to an equivalent postfix expression, that will be stored in 'postfix'. You may assume that 'postfix' is initially empty. This function will return true if we were able to perform the conversion and false if the expression is malformed. Remember, the Shunting Yard algorithm will only fail if the parentheses are mismatched. It is possible for the expression to be invalid, but still pass through the shunting yard.

The only requirement for this function is that you must use a Stack in the implementation. All other details will be left to you. As a suggestion, it might be a good idea to write a function that returns true if a given string represents an operator and another to return the precedence of a given string.

Evaluate Postfix Implementation

You will also need to implement the postfix evaluation algorithm. For this part, you will need to implement the following function prototype:

```
bool evaluatePostfix(const Vector<string>& postfix, double& result)
```

This function will be given the postfix expression produced by the Shunting Yard algorithm as input. It will attempt to evaluate the expression, returning true if evaluation was successful. If evaluation fails (because the postfix expression was invalid), the function will return false instead. The result will be saved in 'result' if the function returns true.

As for `shuntingYard()`, `evaluatePostfix()` **MUST** use a Stack in the implementation. All other details will be left to you. You may write as many auxiliary functions as you'd like, as long as you do not modify the **evaluatePostfix()** function prototype (or **shuntingYard()**) prototypes.

General Advice

Before you begin coding, it would be wise to try running through each of the algorithms on your own. If you cannot execute the algorithm by hand, it is much more difficult to implement it in C++. You should also try converting several simple infix expressions to postfix to make sure you're familiar with the notation. The attached file contains several examples of infix expressions, their corresponding postfix expressions, and the results, so you can use it to validate both your intuition and your code.

Once you feel comfortable with each of the algorithms, it would be a good idea to start working on the `evaluatePostfix()` function, as it is considerably simpler than `shuntingYard()`. It will also give you good practice with working with the Stack class. Get the function working with well-formed expressions before adding error checking. Test this function thoroughly to make sure it does consistently produce the correct answers for valid expressions and that it consistently fails on invalid ones.

After `evaluatePostfix()` has been completed, you should start working on `shuntingYard()`. It would be best to start with simple expressions without parentheses. Once you get the algorithm working with those simple expressions, add the logic for dealing with parentheses. The last addition should be the error checking to deal with mismatched parentheses. As with `evaluatePostfix()`, test `shuntingYard()` thoroughly to make sure it does consistently produce the correct postfix expressions.

As always, do everything you can to make your code "bullet-proof". Try providing inputs that are egregiously wrong to see what your code does, for example. Stress tests like these are helpful for finding weaknesses and vulnerabilities. By eliminating them from the start, your code will be more robust and reliable.

4. Style

Make sure your code adheres to the guidelines provided in the Style Guide (available on Moodle). Your goal is to create code that is concise, descriptive, and easy for other humans to read. Avoid typos, spelling mistakes, or anything else that degrades the aesthetic of your code. Your final submission should be work that you are proud to call your own.

5. Testing:

Test your program to check that it operates correctly for all of the requirements listed above. Also check for the error handling capabilities of the code. Try your program with several input values, and save your testing output in text files for inclusion in your project report.

6. Documentation:

When you have completed your C++ program, write a short report using the project report template describing what the objectives were, what you did, and the status of the program. Does it work properly for all test cases? Are there any known problems? Save this report to be submitted electronically.

7. Project Submission:

In this class, we will be using electronic project submission to make sure that all students hand their programming projects and labs on time, and to perform automatic plagiarism analysis of all programs that are submitted.

When you have completed the tasks above go to Moodle to upload your documentation (a single **.pdf** file), and all C++ program files (**.h** and **.cpp**). Make sure your proof of testing is included in the documentation or is submitted as a separate file. Do NOT upload an executable version of your program.

The dates on your electronic submission will be used to verify that you met the due date above. Late projects will receive **NO** credit. You will receive partial credit for all programs that compile even if they do not meet all program requirements, so make sure to submit something before the due date, even if the project is incomplete.

8. Academic Honesty Statement:

Students are expected to submit their own work on all programming projects, unless group projects have been explicitly assigned. Students are NOT allowed to distribute code to each other, or copy code from another individual or website. Students ARE allowed to use any materials on the class website, or in the textbook, or ask the instructor for assistance.

This course will be using highly effective program comparison software to calculate the similarity of all programs to each other, and to homework assignments from previous semesters. Please do not be tempted to plagiarize from another student.

Violations of the policies above will be reported to the Provost's office and may result in a **ZERO** on the programming project, an **F** in the class, or suspension from the university, depending on the severity of the violation and any history of prior violations.